Module 5: Variables and Outputs

Making Your Infrastructure Configurable and Reusable

Learning Objectives

By the end of this module you will understand:

- How to declare and use variables in Terraform
- Different variable types and when to use them
- Variable validation and constraints
- How to handle sensitive data securely
- Output values and their purpose
- Using `.tfvars` files for environment-specific configurations
- Best practices for parameterization

Discussion: The Problem with Hardcoded Values

Look at this typical Terraform configuration:

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  tags = {
    Environment = "dev"
    Team        = "payments"
  }
}
```

Questions to consider:

- What if you need to deploy to staging? Production?
- What if different regions have different AMIs?
- What if production needs larger instances?

What Are Variables in Terraform?

Variables are input parameters that make your configuration flexible

Think of them like function parameters:

- You define what inputs your configuration accepts
- You can use those inputs throughout your code
- Different values = different infrastructure outcomes

Banking Context: Your bank needs to deploy the same payment processing infrastructure to:

- Development environment (small, cheap instances)
- Staging environment (production-like for testing)
- Production environment (large, redundant, multi-region)

One codebase, different configurations = Variables!

Why This Matters: Variables in Action

Without Variables (Lab 1 approach):

```
# terraform.tf
provider "aws" {
  region = "us-west-1"  # Hardcoded
}

# main.tf
resource "aws_instance" "web" {
  instance_type = "t3.micro"  # Hardcoded
}
```

With Variables (Lab 5 approach):

```
# terraform.tf
provider "aws" {
  region = var.aws_region  # Flexible!
}

# main.tf
resource "aws_instance" "web" {
  instance_type = var.instance_type  # Flexible!
}
```

Benefits:

- Change deployment region → edit one line in terraform.tfvars
- Deploy to different environments → use different .tfvars files
- Prevent mistakes → add validation rules

Variable Declaration Syntax

Basic structure:

```
variable "variable_name" {
  description = "Human-readable description"
  type        = variable_type
  default     = default_value  # optional
  sensitive   = true/false     # optional
  validation {                 # optional
    condition     = <boolean expression>
    error_message = "Error message if validation fails"
  }
}
```

Let's break down each component:

Example 1: Simple String Variable

```
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "us-west-1"
}
```

Using the variable in your code:

```
provider "aws" {
  region = var.aws_region  # Reference the variable
}

resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  tags = {
    Name   = "${var.aws_region}-web-server"
    Region = var.aws_region
  }
}
```

Key points:

- Variables are referenced with `var.variable_name`
- Can be used in string interpolation: `"${var.aws_region}-web-server"`

- If not provided, uses default value

Knowledge Check: Variable Syntax

What is the correct way to reference a variable named `environment` in your Terraform code?

A) `${environment}`

B) `var.environment`

C) `variable.environment`

D) `env.environment`

Required vs Optional Variables

Optional variable (has default):

```
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "us-west-1"  # Terraform uses this if no value provided
}
```

Required variable (no default):

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  # No default - Terraform will prompt if not in terraform.tfvars
}
```

Why This Matters:

- **Optional variables** = convenience (good defaults for common settings)
- **Required variables** = safety (force explicit choices for critical settings)
- Best practice: Use terraform.tfvars to set all values and avoid prompts

---

Variable Types: The Basics

Terraform supports several data types. **Start with these three:**

Type	Description	Example Value
`string`	Text value	`"t3.micro"`, `"us-west-1"`
`number`	Numeric value	`3`, `5.5`
`bool`	True or false	`true`, `false`

Example:

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
}

variable "instance_count" {
  description = "Number of instances"
  type        = number
  default     = 1
}

variable "enable_monitoring" {
  description = "Enable detailed monitoring"
  type        = bool
  default     = false
}
```

Variable Types: Collections

For more complex configurations:

Type	Description	Example Value
`list(type)`	Ordered collection	`["us-east-1a", "us-east-1b"]`
`map(type)`	Key-value pairs	`{dev = "t3.nano", prod = "t3.large"}`

When to use lists:

```
variable "availability_zones" {
  description = "AZs for deployment"
  type        = list(string)
  default     = ["us-west-1a", "us-west-1b"]
}
```

When to use maps:

```
variable "instance_types" {
  description = "Instance type per environment"
  type        = map(string)
  default = {
    dev  = "t3.nano"
    prod = "t3.large"
  }
}

# Use it: var.instance_types["dev"] returns "t3.nano"
```

Variable Validation: Preventing Mistakes

Validation helps catch errors before they cause problems:

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string

  validation {
    condition     = contains(["t3.nano", "t3.micro"], var.instance_type)
    error_message = "Only t3.nano or t3.micro are allowed."
  }
}
```

What happens when you try an invalid value:

```
$ terraform plan
Error: Invalid value for variable
Only t3.nano or t3.micro are allowed.
```

Why This Matters:

- **Prevents cost overruns** - block expensive instance types
- **Enforces standards** - only approved configurations
- **Catches typos** - "t3.mciro" fails validation immediately
- **Banking use case** - ensure compliance (e.g., only PCI-DSS approved instance types)

Knowledge Check: Variable Validation

You want to ensure the `environment` variable only accepts "dev", "staging", or "prod". Which validation is correct?

A) `validation { condition = var.environment in ["dev", "staging", "prod"] }`

B) `validation { condition = contains(["dev", "staging", "prod"], var.environment) }`

C) `validation { condition = var.environment == "dev" || "staging" || "prod" }`

D) `validation { condition = length(var.environment) > 0 }`

Common Validation Functions

`contains()` - Check if value is in a list:

```
validation {
  condition     = contains(["dev", "staging", "prod"], var.environment)
  error_message = "Environment must be dev, staging, or prod."
}
```

`can()` with `regex()` - Pattern matching:

```
validation {
  condition     = can(regex("^t[23]\\.", var.instance_type))
  error_message = "Only t2 and t3 instance types are allowed."
}
```

Comparison operators - Range checking:

```
validation {
  condition     = var.instance_count >= 1 && var.instance_count <= 10
  error_message = "Instance count must be between 1 and 10."
}
```

Banking Context: Use validation to enforce security policies (e.g., production must use multi-AZ, backups enabled).

Sensitive Variables: Protecting Secrets

For passwords, API keys, and other secrets:

```
variable "db_username" {
  description = "Database administrator username"
  type        = string
  sensitive   = false  # Username can be visible
}

variable "db_password" {
  description = "Database administrator password"
  type        = string
  sensitive   = true   # Password will be masked
  # No default - must be provided explicitly
}
```

What `sensitive = true` does:

- Terraform masks the value in plan/apply output
- Shows `(sensitive value)` instead of the actual value
- Prevents accidental exposure in logs and console output

Sensitive Variables: Example

Using sensitive variables:

```
resource "aws_db_instance" "main" {
  identifier = "payment-db"
  engine     = "mysql"

  username = var.db_username
  password = var.db_password  # Value won't appear in output

  tags = {
    Name = "Payment Database"
  }
}
```

What you see during `terraform apply`:

```
# aws_db_instance.main will be created
  + resource "aws_db_instance" "main" {
      + username = "admin"
      + password = (sensitive value)  # Hidden!
    }
```

Why This Matters:

- Prevents passwords appearing in CI/CD logs
- Avoids accidental screen-sharing exposure

- Protects secrets in team environments

Sensitive Variables: Important Security Note

IMPORTANT SECURITY CONSIDERATION:

`sensitive = true` only hides values in Terraform output. The values are:

- **Still stored in plain text in the state file**
- **Visible to anyone with state file access**

Best practices for true security:

1. Use encrypted remote state backends (S3 with encryption)
2. Restrict state file access with IAM policies
3. Consider AWS Secrets Manager or HashiCorp Vault for production
4. Never commit .tfvars files with secrets to Git
5. Add `*.tfvars` to `.gitignore`

Banking Context: PCI-DSS and SOC2 compliance require encrypted state storage and secret management systems.

Knowledge Check: Sensitive Variables

You have a variable for a database password. Which settings should you use?

```
variable "db_password" {
  description = "Database master password"
  type        = string
  # What should go here?
}
```

A) `default = "password123"`

B) `sensitive = true` and no default

C) `validation { condition = length(var.db_password) >= 12 }`

D) Both B and C

Providing Variable Values: terraform.tfvars

Best practice: Use `terraform.tfvars` for all variable values

File: `terraform.tfvars` (automatically loaded)

```
# Default/Dev configuration
aws_region    = "us-west-1"
instance_type = "t3.nano"
```

File: `variables.tf`

```
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "us-west-1"
}

variable "instance_type" {
  description = "EC2 instance type"
  type        = string

  validation {
    condition     = contains(["t3.nano", "t3.micro"], var.instance_type)
    error_message = "Only t3.nano or t3.micro are allowed."
  }
}
```

How it works:

- Terraform automatically loads `terraform.tfvars`
- Values in terraform.tfvars override defaults in variables.tf
- No command-line flags needed

Environment-Specific Configurations

The power of variables: same code, different environments

File: `terraform.tfvars` (dev/default)

```
aws_region    = "us-west-1"
instance_type = "t3.nano"
```

File: `staging.tfvars`

```
aws_region    = "us-west-1"
instance_type = "t3.micro"  # Larger for staging
```

File: `prod.tfvars`

```
aws_region    = "us-east-1"  # Different region
instance_type = "t3.large"    # Much larger for production
```

Deploy to different environments:

```
terraform apply                          # Uses terraform.tfvars (dev)
terraform apply -var-file="staging.tfvars"
terraform apply -var-file="prod.tfvars"
```

Why This Matters: Environment Configuration

Without variables: Copy entire codebase 3 times, manually edit each

```
project-dev/
  main.tf         # instance_type = "t3.nano"
project-staging/
  main.tf         # instance_type = "t3.micro"
project-prod/
  main.tf         # instance_type = "t3.large"
```

Problem: 3x maintenance, 3x testing, hard to keep in sync

With variables: One codebase, different config files

```
project/
  main.tf                   # instance_type = var.instance_type
  terraform.tfvars          # dev: t3.nano
  staging.tfvars            # staging: t3.micro
  prod.tfvars               # prod: t3.large
```

Benefits: Single source of truth, test once, easy updates

Providing Variable Values: All Methods

Method 1: Default values (in variables.tf) - Lowest priority

```
variable "region" {
  default = "us-east-1"
}
```

Method 2: terraform.tfvars (automatically loaded)

```
region = "us-west-1"
```

Method 3: Custom .tfvars files

```
terraform apply -var-file="prod.tfvars"
```

Method 4: Command-line flags - Highest priority

```
terraform apply -var="region=us-west-2"
```

Method 5: Environment variables

```
export TF_VAR_region=us-west-2
terraform apply
```

Variable Precedence Order

When the same variable is set in multiple places, Terraform uses this priority (highest to lowest):

1. **Command-line `-var` and `-var-file` flags** (highest priority, later flags override earlier ones)
2. **`*.auto.tfvars` files** (automatically loaded, processed in alphabetical order)
3. **`terraform.tfvars`** (automatically loaded)
4. **Environment variables** (`TF_VAR_name`)
5. **Default values in variable declarations** (lowest priority)

Example:

```
# variables.tf
variable "instance_type" { default = "t3.nano" }

# terraform.tfvars
instance_type = "t3.micro"
```

```
# Command
terraform apply -var="instance_type=t3.large"
```

Result: Uses `t3.large` (CLI flag has highest priority)

Output Values: Sharing Information

Outputs expose information about your infrastructure:

```
output "instance_public_ip" {
  description = "Public IP address of the web server"
  value       = aws_instance.my_instance.public_ip
}

output "instance_id" {
  description = "EC2 instance ID"
  value       = aws_instance.my_instance.id
}

output "configuration" {
  description = "Configuration used for deployment"
  value = {
    instance_type = var.instance_type
    region        = var.aws_region
  }
}
```

Why use outputs?

1. See important values after `terraform apply`
2. Share data with automation scripts
3. Document important resource attributes
4. Verify which variables were actually used

Output Values: Example from Lab 5

File: outputs.tf

```
output "instance_id" {
  description = "ID of the EC2 instance"
  value       = aws_instance.my_instance.id
}

output "instance_public_ip" {
  description = "Public IP of the instance"
  value       = aws_instance.my_instance.public_ip
}

output "configuration" {
  description = "Instance configuration used"
  value = {
    instance_type = var.instance_type
    region        = var.aws_region
  }
}
```

What you see after `terraform apply`:

```
Outputs:

instance_id = "i-0abc123def456789"
instance_public_ip = "54.123.45.67"
configuration = {
  instance_type = "t3.nano"
  region        = "us-west-1"
}
```

Viewing Outputs

After terraform apply, outputs appear automatically:

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

instance_id = "i-0abc123def456789"
instance_public_ip = "54.123.45.67"
```

Query outputs anytime:

```
# Show all outputs
terraform output

# Show specific output
terraform output instance_public_ip
# Output: "54.123.45.67"

# JSON format (for scripts)
terraform output -json
```

```
# Raw value (no quotes, perfect for scripts)
terraform output -raw instance_public_ip
# Output: 54.123.45.67
```

Using outputs in shell scripts:

```
INSTANCE_IP=$(terraform output -raw instance_public_ip)
ssh ec2-user@$INSTANCE_IP
```

Sensitive Outputs

Protect sensitive information in outputs:

```
output "database_endpoint" {
  description = "RDS database connection endpoint"
  value       = aws_db_instance.main.endpoint
  sensitive   = true
}

output "db_password" {
  description = "Database password"
  value       = var.db_password
  sensitive   = true
}
```

Behavior:

```
$ terraform output
database_endpoint = <sensitive>
db_password = <sensitive>

$ terraform output database_endpoint
<sensitive output hidden>

$ terraform output -raw database_endpoint
payment-db.abc123.us-west-1.rds.amazonaws.com:3306
```

Why This Matters:

- Prevents credentials showing in team demos
- Protects secrets in CI/CD logs
- Still accessible to automation with `-raw` flag

Knowledge Check: Outputs

You want automation scripts to use the instance IP address. What's the best command?

A) `terraform output`

B) `terraform output instance_public_ip`

C) `terraform output -raw instance_public_ip`

D) `terraform output -json`

Best Practices: Variable Naming and Documentation

1. Use descriptive, clear names

```
variable "instance_type" {  # Good - clear purpose
  ...
}

variable "it" {  # Bad - unclear
  ...
}
```

2. Always include descriptions

```
variable "instance_type" {
  description = "EC2 instance type for application servers"
  type        = string
}
```

3. Group related variables

```
# Network variables
variable "vpc_cidr" { ... }
variable "subnet_cidrs" { ... }

# Compute variables
variable "instance_type" { ... }
variable "instance_count" { ... }
```

Best Practices: Defaults and Security

4. Provide safe defaults when appropriate

```
variable "enable_monitoring" {
  description = "Enable detailed CloudWatch monitoring"
  type        = bool
  default     = true  # Safe default - more monitoring is better
}
```

5. No defaults for critical or sensitive values

```
variable "db_password" {
  description = "Database password"
  type        = string
  sensitive   = true
  # NO DEFAULT - must be explicitly provided
}
```

6. Use validation to prevent mistakes

```
variable "environment" {
  description = "Deployment environment"
  type        = string

  validation {
    condition     = contains(["dev", "staging", "prod"], var.environment)
    error_message = "Environment must be dev, staging, or prod."
  }
}
```

Key Takeaways

1. **Variables make infrastructure configurable** - same code, different environments
2. **Start simple** - use string, number, bool types first
3. **Validate inputs** - prevent misconfigurations early with validation rules
4. **Mark sensitive data** - protect secrets from exposure
5. **Use .tfvars files** - separate configuration from code
6. **Outputs share information** - critical for automation and documentation
7. **Never commit secrets** - use .gitignore and secret management tools
8. **Document everything** - descriptions help team members understand intent